Computer and Automation Institute,
Hungarian Academy of Sciences

KIBE

CHARACTER HANDLING

Budapest 1980

Készült

a Magyar Tudományos Akadémia Számitástechnikai és
Automatizálási Kutató Intézetében

Felelős kiadó:

DR VÁMOS TIBOR

ISSN 0209 - 4029

8011892 MTA KESZ Sokszorosító, Budapest. F. v.: dr. Héczey Lászlóné

C O N T E N T S

Ø. INTRODUCTORY REMARKS

"Kibe" is a system of services, defined in a machine-independent way, intended for (possibly interconnected) mini - or microcomputers. It is to handle "character devices", where character device may mean, besides "genuine" character peripherals, anything that allows for byte-wise data transfer, e.g., a plotter, a data line, or parts of memory, real or virtual.

Its properties: -

(a) Devices can be added or removed at any time.

(b) Devices, if physically capable of asynchronous operation, can be operated concurrently with each other and with program run.

(c) Conflicting device requests by different (possibly independent) processes are arbitrated, and get seviced in (reorderable) order.

(d) Except for routines performing immediate physical input or output, device handling is logically not dependent on the devices and machine actually used.

(e) Device handling can be embedded in a virtual data and program handling régime. (It *is* in fact embedded into one of them, that of "Books"; see "Motor, a Mini-Micro Blow-Up", Budapest 1979. It should not be very hard to conform with other disciplines.)

(f) Characters can be processed both in device code and in a device-independent internal character representation. The later one allows for data ordering, syntactic analysis, logical device control, etc.

(g) Syntactic primitives (integer, floating-point number, name etc.) can be made understood, and converted between "external" and "internal" representation; "formatted" conversions are provided for (where "reading" *produces* a "format", and "writing" *is governed* by it).

(h) "Higher level" character display is definable to the extent that today's more sophisticated computer peripherals (e.g., graphical displays, plotters) can make use of it. These facilities can be enhanced to produce a veritable "typographical" processor; this, however, is not aimed at. Instead, an interface to existing typographical processors (e.g., D.E. Knuth's "TEX") is proposed.

(i) Device handling can (and does) serve as part of higher--level languages.

*

"Kibe" has not its English meaning; it is Hungarian for "out-in".

1. DEVICES

A Device is anything enabling byte-wise transfer. It is considered to be "outside"; "inside" being (real or virtual) memory. Transfer from a Device is called "reading"; to it, "writing". A physical device allowing for both will be considered two Devices.

An "ideal" Device is a Device capable of everything that can be prescribed in terms of the subsequent specification. A "real" Device is something that exists. Its capabilities may be restricted. If requested to do something it is not capable of, some sensible default action is taken. (E.g., if there are no Italics, Roman letters appear; if no lower case, upper case; if no colours, then so it is.) Whether, and which, default actions are taken tacitly, or are accompanied by some kind of warning or notification, rests with the actual implementation.

A "fictive" Device is a part of real or virtual memory *used as* a Device. (E.g., "printing" onto a virtual printer file on disc.) Fictive Devices are "ideal" by definition.

A Handler can control up to 255 Devices at a time. (The stock of Devices under Handler control can be changed at any time of Handler inactivity.) Devices are identified by *numbers* between 1 and 255.

DISCIPLINARY SUGGESTIONS

Transfer organization can be conceived in many ways. At this deepest level, as much as possible has been purposely left undecided. In most practical cases, however, this level is too low to serve as a framework for programming. Thus an operational "system environment" could be desirable, with the following properties.

There are Device *types*, coded by numbers or names, or both; Devices of the same type are identified by type-and-number; it is the system's task to find out the corresponding physical device. Programs can ask (say, by number = Ø) for *any* Device of a type. If there are terminals, local devices can be identified in a terminal-independent way; it is again the system's task to find the correspondence.

Programs may inquire whether Devices are available or not. And, hopefully, there will be some sort of standard Device description, to be consulted by programs. (E.g., paged or not; if yes, page or screen sizes; erase or backspace or overprint possible or not; character sizes available; etc.)

Some Device properties may be program-definable. (E.g., size and location of a message window on a screen.) Such cases can be dealt with by either "generating" a separate Device or appropriately manipulating a fixed Device. Programs need not be aware of which solution has been chosen.

2. CHARACTERS

Characters can be handled as coded by the particular Devices (that is, as uninterpreted byte patterns). In addition, there is a Device-independent "internal" character code, called simply the Code. Device Drivers convert from/to Code or transfer unchanged patterns as requested.

The Code has 256 characters, code-numbered :ØØ-:FF. 128 of them, :ØØ-:7F, have corresponding symbols. There are eight possible set of symbols, the choice between which is made by "switches". The rest are the "control" characters.

Not necessarily all character codes need be produced by a reading Device or understood by a writing Device. It is the Device Driver's decision to replace non-existent symbols by similar ones, or multisymbols, or error marks (code number :8Ø is suggested on reading, the question mark on writing), or nothing at all.

Symbols are in fact arbitrary and can be arbitrarily changed. In case of a change (other signs, "national" alphabets, etc.) the only thing to be taken in account is that whenever a *sense* is attached to a character by the Handler (e.g., "digit", or "letter", or "decimal point", or "quote mark"), this sense is attached to the character *with that code number* (whatever its chosen visible form). These correspondences can be altered, too; but not without altering the Handler itself.

Symbol characters; set Ø:

1. Space
:ØØ - ␣
2. Bar
:Ø1 - _
3. Digits
:Ø2-:ØB - Ø-9

4. Letters

:ØC-:25 - upper case

:26-:3F - lower case

5. Signs

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	¢	$	&	%	@	#	!	?	'	"	.	:	;	,	-	|
5	(	)	[	]	{	}	⟨	⟩	∈	∉	⊂	⊆	⊃	⊇	≡	≡
6	=	≠	<	≤	>	≥	+	−	±	×	*	•	÷	/	\	¬
7	~	∘	○	□	√	∫	∂	Σ	Π	∨	∧	∪	∩	—	|	blk

The "bar" is the name-concatenator as used, e.g., in PL/1. - If a "$_{1o}$" symbol exists, it replaces "@" (:44). - :4E is the "hyphen", as distinguished from :67, "minus". - :7D and :7E join together to produce contiguous lines. - :7F, "blk" (blank) is to correspond to the "null" character of the Device, if any (e.g., blank tape on paper tape, or filler in an extraneous buffer); it is never produced at reading (the Device Driver suppresses it), and is effectless at writing on Devices without a "null" character (e.g., card punch).

Set 1 is for Greek, Cyrillic, and additional Latin symbols (accented or special letter shapes), in this order. (Greek is meant unaccented; separate accents and accented letters give more than a hundred combinations, so if full Greek is needed, this can go to another set.)

Set 2 is for Math formula building.

Sets 3-6 are implementation-defined; the may contain other alphabets (e.g., Katakana) or types (e.g., fraktur or script), special symbols, symbols in upper and lower index position, who knows. (Right-to-left or vertical setting can be achieved by changing

the implied positioning, see below; character-wise positioning is always feasible of course.)

Set 7 is for line-drawing, to be described elsewhere.

Before setting forth control character interpretation, Device graphics must be explained.

A writing Device, if driven in Code, has "graphics".

There is a "vertical unit" and a "horizontal unit"; they are arbitrary and need not be equal.

A symbol character has a "face": a quadrangle including its visible shape, the height and width of which are expressible in units.

Symbol sets come in different "sizes". Within the same size, all face heights are supposed equal; widths not necessarily.

A non-negative amount of units of "air" (empty space) can be added on bottom of faces: "line-spacing"; similarly to the right of faces: "letter-spacing".

The symbol's "body" is the face with line-spacing and letter--spacing added.

There is a "frame" (or "screen", or "window", or "page"): a quadrangle having as reference point the upper left corner, and extending rightwards and downwards by a limited or unlimited number of units. (If limited, it is expected not to exceed 255 units in either direction.)

The "position" of a point in the frame is its vertical and horizontal distance from the upper left corner, measured in units. The position of a symbol is the position of the upper left corner of its face (= of its body).

There is a "current position": whenever a symbol is to be written, it gets located at that position.

There is a "direction" of writing. At writing a symbol, the current position is stepped, as required by the body and the direction, to locate the next symbol. Frame overrun is handled in the obvious way.

Within a "block" of equally directed, equally sized and equally line-spaced symbols, the "place" of a symbol is defined as follows: the place is m,n if it is in the m^{th} line from top and in the n^{th} symbol location from left, the numbering starts from Ø. (This is self-explanatory for lines, and so it is within lines provided body widths are equal or lines are full. Are they not, then fill unwritten parts of the line with "letter-n-wide" spaces, remainders neglected, and count in this line.)

The "state" of the Device is described in a six-byte (48-bit) quantity.

bits Ø-7: Current vertical position.
bits 8-15: Current horizontal position.
bits 16-23: Writing direction, in 3 degrees counterclockwise, Ø is left-to-right. Leftmost bit clear requires symbol shape rotation; if set, symbols remain standing.
bits 24-26: Symbol set.
bits 27-31: Symbol size. Not in units, but style numbers, ordered to ascending magnitude in height, and within height in width.
bit 32: Line-spacing, (approx.) a fifth of face height.
bit 33: Letter-spacing, (approx.) a fifth of (average) face width.
bit 34: Italic. (With line-drawing: dotted.)
bit 35: Underlined. (With line-drawing: dashed.)
bit 36: Bold (bright, thick).
bits 37-39: Colour; Ø black, 1 white, then in increasing wavelength.

bit 4Ø: Clear means overprint, that is, previous symbols remain visible even if new symbols appear in the same location. If set, the overwritten symbol is cleared away; this is possible only if concurring symbols have the same definite place.
bit 41: For set, a cursor is to be shown to mark the current position.
bits 42-47: Freely definable for special usage.

Initial values are all zeros except for symbol size which is 8.

Now control characters are: -

:8Ø - UD. The "undefined" mark.
:81-:86 - VP,HP,DI,SL,SM,SN. Replace the state bytes Ø-5, resp., by the byte that follows (the "next character"). Too big VP, HP values yield max.
:87-:88 - VR, HR. Relative positioning: the following byte gets added to the current vertical and horizontal position, resp.; over- and underflown frame limits result in max. and min. position.
:EØ - HI. Rise, vertically to direction, by cca. $\frac{3}{5}$ face height.
:E1 - LO. Lower " " " " " " "
:E2 - SR. Step, in direction, by cca. $\frac{1}{5}$ face width.
:E3 - TR. " " " " 1 unit.
:E4 - IS. Increase symbols to next larger standard size.
:E5 - DS. Decrease symbols to next smaller standard size.
:E6-:E9 - S3,S2,S1,SØ. Set symbol set number (bits 24-26) to 3,2,1,Ø, resp.
:EA - LS. Set letter-spacing bit (bit 33).
:EB - IT. Set Italic bit (bit 34).
:EC - UL. Set underline bit (bit 35).
:ED - BD. Set bold bit (bit 36).
:EE - NM. Reset bits 33-36 to zero.
:EF - RS. Reset the whole state to initial values.
:FØ - BØ. Mode Ø (some button with no graphical interpretation).
:F1 - B1. Mode 1 (" " " " ").
:F2 - TB. Tabulate.

:F3 - CI. Insert character.
:F4 - CD. Delete character.
:F5 - LI. Insert line.
:F6 - LD. Delete line.
:F7 - RG (or →). Step to next place right (cyclically).
:F8 - UP (or ↑). " " " " up " .
:F9 - LF (or ←). " " " " left " .
:FA - DN (or ↓). " " " " down " .
:FB - HM (or ↖). Home (current position to Ø, Ø).
:FC - PG (or ↘). Erase (home + clear), or new page.
:FD - NL (or ↙). New line.
:FE - ET (or ↗). End of text when reading.
:FF - DL. Delete, or end of text when writing.

Characters :89-:DF are intended for extensions - e.g., graphic routine calls, for handling blocks of symbols and defining complex actions (such as blinking, moving). This goes beyond strict "character handling" but can be built upon it to a certain extent. Characters :89-:BF are supposed to carry a "follower byte", characters :CØ-:DF not.

3. TRANSFERS

Transfer is between a Device and a part of (real or virtual) memory. Calls to transfer operations refer to a Request Descriptor, REQDES, which must be in real memory (core store). A Request Descriptor must not be used in calls for different Devices at the same time.

The (address of the) Request Descriptor "identifies" the "Claimant": requests with the same Request Descriptor count as issued by the same Claimant.

A Device is either "idle" or is "owned" by a Claimant. The Owner is free to use the Device (write or read on it). Other Claimants for that Device are ordered in a queue (of unrestricted length). As soon as the current Owner releases the Device, the Claimant next-in-queue becomes Owner (if none, the Device turns to idle).

The queue can be joined by a Claimant
- as last in queue: NEW,
- as next to the current Owner, jumping all others: NXT,
- taking over from the Owner, who will continue "next": NOW.

For an idle Device, these are of course equivalent. Iterated calls are permitted; they can improve, but not deteriorate, the Claimant's position.

NOW is for desperate emergency measures. At collision time, a character may be lost (which belongs either to the suspended or the suspending Claimant, depending on hardware functioning).

NEW need not be separately given; it is automatically performed by any read/write call when needed.

A claim call returns immediately (that is, the Claimant will be queued but is not waited to become Owner).

Releasing: RLS.

Releasing is immediate, a possibly uncompleted transfer will be discontinued. Releasing of a non-Owner simply waives the claim. Releasing non-existent claims is harmless.

Some transfer modes perform releasing automatically for certain characters (see below).

"Transfer" means writing or reading, according to the Device.

A call for transfer can be given at any time (the caller is not bound to be Owner); transfer will be performed in due course. Transfer termination can be found out either by consulting REQDES or by making the Handler issue a routine call when finished. After termination, transfer can be repeatedly called for, and will be processed in succession, as long as the Device stays unreleased (and nobody else's NOW butts in). A new call that precedes termination overrides the old one; that is, replaces it if not even started, breaks it at some point if under execution.

Transfer can be single-byte or multi-byte. Multi-byte transfer gets terminated if either a termination condition (different for different transfer modes) is met or the memory part specified in the call is exhausted. Memory can be real or virtual for multi-byte; only real for single-byte.

Transfer can be made with conversion from/to Code or without it. If in Code, an ET character on reading and a DL character on writing causes release. (Release implies of course termination.) The character causing release comes in on reading but does not go out on writing.

Transfer modes are as follows.

Without conversion:

BYTE - single-byte.
SEND - multi-byte.
STRG - multi-byte, termination on :FF.

In Code:

CHAR - single-byte.
MESS - multi-byte, termination on DL.
CLAU - multi-byte, termination on any control character.

For any reading Device, a writing Device can be assigned as "Echo Device". This assignment is installation-dependent (and changeable at any time when the affected Devices are inactive). Reading with ECHO mode causes the bytes received immediately written out on the Echo Device (unconverted or converted both ways, as the case is). Terminators read in STRG, MESS and CLAU mode and control characters read in CHAR mode are not echoed.

An Echo request operates the Echo Device without queueing for it. (To have its monopoly, it can be claimed for separately.)

Possible error indications: The Device (a) does not exist, (b) is off-line. (Particular Device Drivers are free to specify other types of error.) A request for an unassigned or non-existent or off-line Echo Device produces no error indication (nor an echo).

REQDES

Ø : DN
1 : ST
2-3 : LK
4-5 : ML
6-7 : LG
8-9 : UR

DN is the Device number.

ST has this layout: E,T,O,M_1,M_2,I_1,I_2,I_3

The three leftmost bits carry the state information given by the Handler, their initial contents are irrelevant.

E : set if error - if no such Device, DN will be zeroed; if off line, DN remains

T : set if transfer is finished

O : set if the Claimant is Owner

The remaining five bits contain the caller's request.

With claiming-releasing requests: -

M : ØØ - RLS
Ø1 - NEW
1Ø - NXT
11 - NOW

I : ØØ1 for all

At return, request bits will be cleared.

With transfer requests: -

M_1 : set wishes for user routine call at termination

M_2 : set requires ECHO mode

I : Ø1Ø - BYTE
Ø11 - CHAR
1ØØ - SEND
1Ø1 - MESS
11Ø - STRG
111 - CLAU

At return, request bits are unchanged.

(Patterns with I : ØØØ are unused.)

LK is a link, set by the Handler. It points to the REQDES next in queue; zero if none. Initial contents are irrelevant.

ML specifies the memory location transfer is to be performed with. If transfer is single-byte or if the leftmost bit of LG is clear, this is a core storage address; otherwise an information identifying a virtual memory location (in case of Motor: a Parcel reference, with length 1, to a Book, preferably sequentially handled). Single-byte transfers leave ML unchanged; multi-byte transfers step up either ML itself or the virtual location pointer, as the case is, to point to the location immediately following the last-transferred bit. (If the Device has been found off-line, the location pointer will be increased by one on writing, unchanged on reading.)

LG is relevant only with multi-byte transfer. Leftmost bit set means transfer with virtual memory. The rest is the "counter"; it is decreased by one for each byte transferred; decreasing to zero causes termination (after the transfer). A zero *initial* value causes no termination and remains unchanged ("infinite size").

UR - User routine address. It will be called if a transfer request with M_1 set is finished. The routine is given two quantities: the Device number, D, and the REQDES address, R. It is free to do anything it wants (including calls to the Handler).

ML, LG, UR are irrelevant and unchanged for claiming-releasing requests.

DISCIPLINARY SUGGESTIONS

Parallelling, interrupting etc. intricacies, if needed at all, have to be implemented somewhere deep, since less pretentious functions can be built on those, but not the other way round. Yet everyday programming practice is, at least nowadays, rarely concerned with synchronizing etc. subtleties, and rightly so if problems are not those of inner-program organization but of co--existence with unknown foreigners. And, e.g., when in need of data, low-level programs have in many cases, high-level programs in most cases no application whatever for the time in-between. Thus, what should be linguistically simple is the extremities: Simple "waited" read/write in the Fortran style, and starting a "parallel" transfer of a whole "file". Add perhaps (in languages equipped with "on-conditions") as third "primitive" this: "on" depressing some button do start a procedure. - If there is multi-tasking, it may find use for "waiting time" (getting information from the calls themselves).

Some characters have no visible shape (end-of-text, newline, etc.), which is inconvenient in writing character constants in the program. There are two plausible remedies: (a) to permit a free redefinition of symbol code-numbers within the program; (b) to predefine certain visible symbols as *meaning* other characters and to define a symbol (say, "!") as "escape" character that restores the character following it to its original meaning (the escape character itself being writable by doubling).

In multi-tasking, a clean-up procedure is required to release Devices on behalf of broken-down Claimants.

A more elaborate priority system can be based on the above primitives (this need not be any deeper). Still, there is, notwithstanding the propaganda to the contrary, no point in having a very refined priority system for logical Device handling.

4, DRIVERS

Every Device has a Driver. (A Driver can be had by several Devices.)

The interface between a Driver and the Handler is this.

The Handler contains a Device description table, DEVDES, consisting of 16-byte Device Descriptors for each Device, in sequence of the Device numbers, the first being that for number 1.

A Device Descriptor looks like this.

```
Ø - 1 : DRIVER
    2 : IO
    3 : MODE
4 - 5 : OWNER
6 - 15: WORK
```

DRIVER contains the starting address of the Driver routine; zero if no Driver (or no Device).

IO contains Ø for a writing Device; the number of the Echo Device for a reading Device - :FF if no Echo.

MODE is for communication of the Handler with itself and the Driver.

bit Ø : set by the Driver in case of failure (Device is off--line);
bit 1 : set by the Handler if the Device is to be driven in "wait mode" (by polling, without interrupt);
bit 2 : set by the Handler, if the Driver has to convert from/to Code;
bit 3 : unused;
bit 4 : remembers the Handler that an echo character is being written;

bit 5 : remembers the Handler that a release character is being written;
bit 6 : remembers the Handler that a terminator character is being written;
bit 7 : remembers the Handler that a last character is being written.

OWNER is pointed by the Handler to the Owner's REQDES; zero for an idle Device.

WORK is for the Driver's purposes.

Two additional "variables" get passed in each call on the Driver: DX and BC. At call, the Driver receives in DX the Device number multiplied by sixteen.

The transferred byte is given/taken in BC.

When handling by way of interrupt, after starting the Device, the Driver is to make a normal "return". - The routine activated by the interrupt shall use the Handler's SAVE procedure to start with.

Whether interrupted or not, when finished, jump to DONE.

5. ITEMS

Items are syntactic units. They have *Shape* ("outer") representations and *Value* ("inner") representations. There are Handler operations to do the conversion: conversion from Shape to Value will be called *In;* from Value to Shape, *Ex.*

Items have *Terminators*. At In, the Terminator is the first character that cannot belong to the Item; at Ex, it is specified by the caller.

Items have *Formats*. At In, a Format is produced by the Handler; at Ex, the Handler is controlled by a Format.

(A Shape uniquely determines a Value and a Format, and conversely. Exing data as they have been Inned results in a "semantically" - but not necessarily formally - equivalent Shape.)

Item types are: *Txt, Nam, Hex, Int, Frc, Scl, Void.* (See below.)

The size of the Value representation is double-word (4 bytes) for Int, Frc, and Scl; any number of double-words for Txt, Nam and Hex; Void has no size (and no Value).

Terminator and Format take a byte each.

Item handling is performed by the Handler routines INSYNT or INITEM (doing In) and EXITEM (doing Ex).

The Shape representation is expected/produced in (real or virtual) memory, in Code ("internal" character representation, see Sect. 2).

Routine arguments are: -

LOC (2 bytes). - To be pointed, at call, to the first character position of the representation to be processed or produced, or, in case of virtual memory, to an appropriate virtual reference (as in ML with Transfers, see Sect. 3). At return, it or the virtual reference will point to the first position following the Shape representation (the one after the Terminator). With Ex, the position pointed to will contain a DL (:FF).

VALUE (4 bytes). - Takes the Value representation. Conversion of a Txt, Nam or Hex exceeding a double-word is made by iterated calls; completion or otherwise is shown by the Format. For Void, VALUE is unchanged or irrelevant, resp.

FORMAT (1 byte). - The Format. Note that some Format patterns (ØØØØØ11Ø for Txt, ØØ1XX111 for Nam, ØØØØ1111 for Hex, see below) are "continuation" Formats, declaring that conversion is not yet complete.

TERM (1 byte). - The Terminator; in case of uncompleted In, it contains the "next" character; the next call, with unchanged TERM and FORMAT, continues conversion. At Ex, no Terminator goes out if either the Format is a continuation Format or TERM is DL (:FF). (For continued Ex, any kind of "no Terminator" is good enough; TERM need not contain the information it had at In.)

When starting In, Format is to be cleared, or at least set to a non-continuation pattern. (More accurately: For a continuation pattern, In works as if continuing for the given Item type. So this can be purposely used, e.g., for prefixing an Item, or to make the Handler accept a Nam starting with a digit, a Txt without a quote or a Hex without a colon.)

INSYNT has an additional argument SYNTAX (2 bytes), to specify syntactical variants (see below).

Item types:

Txt

Shape at In:

A - possibly empty - sequence of characters enclosed between quotes (apostrophes). The Terminator is the first character after the closing quote. ET and DL (:FE and :FF) are immediate Terminators (even without a closing quote). There is an "escape" character, "!", qualifying the next character an "ordinary" character. (So a quote preceded by an "!" becomes part of the sequence instead of delimiting it. To put in an exclamation mark, give two of them.) The escape character does not override immediate Terminators.

Value:

The sequence of character code numbers (without the including quotes), one byte each; the double-word, if not full, padded on the right with zero bytes.

Shape at Ex:

The character sequence (without quotes). An :8Ø gets suppressed (but counts as a character), an :FF causes discontinuation (no more characters and no Terminator will be Exed).

Format: ØØØØ Øddd, $1 \leq d \leq 6$

d is the number of characters plus one if completed (1 for empty, 5 for full) and is 6 if uncompleted.

Nam

Shape at In:

An alphanumeric character sequence starting with a letter. Letters may be all upper case, all lower case or first letter upper case and the rest lower case (any mixing of letter cases will be interpreted this way).

Value:

Replace the lower case character code numbers by those of the corresponding upper case characters. Cut the resulting sequence into slices of six, the last being possibly shorter. Each double-word holds the Value for a slice in order, the Value being defined thus. Denote the slice $c_1 c_2 \ldots c_k$, $k \leq 6$; let b denote 37 (the number of possible symbols: the letters, the digits, and - if permitted - the bar). Then the Value is

$$\sum_{i=1}^{k} (c_i - 1) \frac{b^{6-i} - 1}{b - 1} + k + 165797Ø7Ø9.$$ - Note that this representation orders the Nams lexicographically (if compared as unsigned numbers). - Not all possible Value patterns are produced by In. So no Nam starts with a digit, and, moreover, no Nam (nor Nam-slice) has Value 165797Ø7Ø9 or less (giving space for internally generated Nams).

Shape at Ex:

The same as with In. Nams starting with a digit appear as such, value 165797Ø7Ø9 yields the "empty Nam" (consisting of no character), Values less than this appear as (decimal) numbers.

Format: ØØ1ℓ uddd

d is the number of characters (1 to 6) or 7 for uncompleted. At Ex, if d is greater than needed, the rest at right will be padded with spaces (7 counts as 6); if d is less than needed, still the full Nam appears.

u = 1 : an upper case letter, ℓ = 1 : a lower case letter has occurred. (Digits and the bar are neither upper nor lower case.) At Ex, ℓu = Ø1 is all upper case, 1Ø is all lower case, 11 is first character upper case and the rest lower case, ØØ if untrue works as Ø1.

Notes. (a) Such a representation, with b ≤ 4Ø, fits Nams of up to six characters into a double-word provided the set of characters Nams are composed of does not exceed 4Ø symbols (and then there remain 93941655 patterns). This can be used to represent other symbol sets. (b) The remaining patterns could be used to define an additional family of Nams instead of appearing as decimals. This can be easily added to, but is not contained in, the Handler.

Hex

Shape at In:

A sedecimal number prefixed by a colon. A-F can be written a-f. An empty Hex counts as Ø.

Value:

The corresponding binary.

Shape at Ex:

The numbers, unprefixed. A-F upper case.

Format: ØØØØ dddd, 7 ≤ d ≤ 15

d is the number of digits plus 6, or 15 for uncompleted. At Ex, if greater than needed, leading zeros are output (d = 15 counts as 14), if less, still all significant digits appear. Zero is considered as having 1 significant digit.

Int

Shape at In:

A possibly signed decimal integer in the range $\left[-(2^{31}-1),\ 2^{31}-1\right]$; insignificant leading zeros may precede.

Value:

The corresponding two's complement binary number. For integers outside the range, signed max. abs. value ("machine infinity") is taken. The pattern :8ØØØØØØØ (corresponding to -2^{31}) is produced by no In (and may serve as "nil").

Shape at Ex:

The integer, signed if negative. :8ØØØØØØØ appears as -2147483648.

Format: ØØØ1 dddd

d is the number of significant digits minus 1; Ø for zero. At Ex, d+1 is the field width; if greater than needed, the number appears right-justified, otherwise contiguously (with no spaces before and after).

Frc

Shape at In:

A possibly signed decimal fractional number, that is, a number containing a decimal point; both integer and fractional part may be empty, and superfluous zeros may precede. The number of significant digits must not exceed 127. (Even the best 4-byte floating-point machine representations have precisions where the eighth decimal digit is hardly significant, so it makes no sense, arithmetically, to have more.)

Value:

The machine's floating point number representation. If to big, signed max. abs. value ("machine infinity"); if too small, zero. - Zuse ("PDP") representation (cf., e.g., "Floating Point Algorithms, as used at the ... Computer and Automation Institute, Hung. Acad. of Sci.", Budapest 1979) is considered "standard"; where machine representation is not standard, conversion routines shall be available. The Value corresponding to standard :8ØØØØØØØ has the same role as with Int.

Shape at Ex:

The number, signed if negative. If abs. less than 1, one zero appears before the point. - If not acceptably representable as Frc, it will appear in a corresponding Scl shape. :8ØØØØØØØ yields minus-zero.

Format: Ø1fg hddd

At In: f,g,h is Ø; d is the number of significant digits minus 1; for zero, d is the number of fractional zero digits the number is written with; but d does not exceed 7.

At Ex: f = Ø. Then d has the meaning as with In; rounding is done at the corresponding place. gh = ØØ results in contiguous output, 1Ø left-justifies the number in a field of width d+7,

Ø1 right-justifies it in a field of the same size, 11 centers it around the decimal point in a field of width 2d+7. If the number is so big that significant digits don't "reach" the decimal point or so small that they don't fit into the field because of too many leading fractional zero digits, the number will appear as Scl, with the same d, and k = d for big, k = Ø for small, appropriately fitted in the field. - f = 1. Then d is the number of fractional digits the number is to be written with; rounding is to this place. For negative numbers rounding to zero, the minus sign is not suppressed. h = Ø is contiguous output, 1 wants right-justification in a field of width 1Ø. The number of digits in the integer part is at most 8-d; bigger appears as Scl with k = d = 3, j = h.

Scl

Shape at In:

A number in Int or Frc form followed by an exponent symbol followed by a possibly signed exponent. The exponent symbol may be "@" (or "$_{10}$") or (if permitted) "E" or "D".

Value:

As with Frc.

Shape at Ex:

The number, signed if negative. Exponent symbol is "@" (or "$_{10}$"); the exponent is two digits, always signed (plus for zero).

Format: 1jkk kddd

d is the number of significant digits minus 1; for zero, d is the number of fractional zero digits the number is written with. At In, if 7 is exceeded, d still will be 7. At Ex, rounding is done at the corresponding place.

k is the number of significant digits preceding the decimal point minus 1. In is permitted outside the range of k; k will be set minimal or maximal, as the case is. At Ex, if k exceeds d, the excess gets neglected; for a zero number, k is considered Ø.

j is Ø at In. At Ex, j ≠ Ø results in contiguous output, 1 right-justifies in the field, field width is d+7.

Void

No Shape, no Value. Means simply that no other Item has been found, or is to be put, before the Terminator.

Format: ØØØØ ØØØØ

Note. Format patterns ØØØ1 11xx (in Int) and Ø111 xxxx (in Frc) can be freed for extensions. (Scl with k > d is not very helpful, either; but these are not handy patterns.)

Syntactical variants

Only In is affected. Call INSYNT (instead of INITEM); bits of SYNTAX, from left to right, are interpreted the following way if set.

Ø : No Txt accepted.
1 : No Nam accepted.
2 : No Hex accepted.
3 : No Int accepted. - If Frc is permitted, Int will convert as Frc.
4 : No Frc-Scl accepted. - If neither Int nor Frc-Scl are accepted, a Nam may start with a digit.
5 : Control character suppression, except ET and DL.
6 : Newline and Page suppression.

7 : Space suppression, except in Txt.
8 : Comment suppression; if a "¢" (:4Ø) is encountered, it and the following characters will be suppressed until and including the next "¢" or until and excluding the next semicolon or the next unsuppressed control character (whichever comes first).
9 : Txt is enclosed between ditto signs (double quotes) instead of apostrophes (single quotes).
10 : Txt is immediately terminated by any (unsuppressed and unescaped) control character.
11 : Txt is immediately terminated by any (unescaped) semicolon.
12 : A "bar" (:Ø1) is accepted as part of Nam.
13 : Hex is prefixed by "#" (rather than ":").
14 : "E" and "D" are, too, accepted as exponent symbols.
15 : No signed numbers. (Signs preceding numbers are taken separate entities.)

Clearly inhibiting all Item types but one amounts to prescribe the Format; inhibiting all of them makes all Items Void (and makes the characters come in one by one as Terminators).

No special syntax is given for fixed-position (unterminated) input since it is easily reducible to forms handled here.

Octals as used, e.g., in PDP word display can be handled at In and Ex as Ints or Hexes, with internal conversion of the Number (the same applies of course to any base <1Ø or <16, resp.)

DISCIPLINARY SUGGESTIONS

Device and Item handling being now cleanly separated, it is time to point out that what a normal programmer wants is immediate *Item-wise* handling of *Devices*. So the next thing to do is to undo things done.

Secondly, some kind of "Format language" would be desirable. Such a language should admit of symbolic specifications for general syntax, and for Format handling. Something in this style is meant (L stands for letters, m and n for numbers):

```
&n - Txt;
$L:n - Nam;
:n - Hex;
#n - Int;
%L:n - Frc;
@L:m:n - Scl.
```

Add to this:

```
=    - single character (this is essentially the Void);
/m:n - relative positioning;
\m:n - absolute positioning;
?    - skip (next Item in stream to be left out);
!    - substitute (next Format to be taken from next place in
       the stream).
```

Maybe a specifier like "[procname]" would be useful: routine call at the time it is encountered.

A "Format sentence" is a string of such Format specifiers, separated by commas. Parantheses, repetition factors - n*... - may occur.

Specifier and sentence variables are permitted.

(A rudimentary system of this type is contained in "Kalap, a Batch Monitor System", Budapest 1978, Sect. 3.2; evaluation rules could be obtained by generalizing the rules given there.)

6. SIZE, SPEED

Data apply to TPA/7Ø, which can be considered representative of good minicomputers. (PDP/11 code is somewhat longer and somewhat faster.)

The Handler is about 23Ø words plus Device Descriptors (presently 8Ø words). The Driver (DOIT) is about 18Ø words, cca. 1ØØ of which are for conversion routines and tables. Item is some 8ØØ words. Practically all of it can be placed in virtual memory without considerable slow-down.

Transfer management time for a character with conversion is about 1 msec in the worst case, very much less on average. This has been felt satisfactory as compared with present peripheral speed. (There are longer but much quicker algorithms, and dedicated hardware could also be used to speed up. No need for this, at present. - To make conversion time and hardware time simultaneous for the *same* Device is not worth the effort.) Item requires between $\frac{1}{3}$ - 3 msec in most cases; very large or small floating-point numbers take more, simple things take less.

*

The general moral is: If conversion is needed at all, do it at points where characters enter and leave the system, and *not* while processing things within a program nor when communicating through a high-speed channel within the system.

7. ALTERNATIVE IMPLEMENTATION

(a) The Claimant provides no REQDES resident in memory but simply tells his wishes through a medium that may vanish after call; it is the Handler which provides space to retain information (out of a pool of, say, 8-byte cells). The Claimant has separate instructions to examine the state.

(b) The Claimant is identified, not by a Request Descriptor, but by a "Task Number" (which may, or may not, co-incide with his identification under an overall task discipline). Requests with the same Task Number are processed contiguously.

Advantage: Multitasking is easier (requests of tasks not present in core, etc.). Disadvantage: The pool can run out, so there are additional complications.

This can be pushed even further. The whole transfer can be made in a Handler pool (and not in caller-given areas).Callers communicate, without being aware of it, with this pool instead of Devices.

Things like this can be built *upon* the previous routines. But maybe it is less of a loss and more of a gain to have them on the *deepest* level.

KIBE ON TPA/7Ø

Devices already driven are:

1: console (alphanumeric display) input (echo:2),
2: " " output,
3: paper tape input (echo:1Ø),
4: " " output,
5: teletype input (echo:6),
6: " output,
7: graph. display alphanumeric keyboard input (echo:2),
8: graph. plotter output,
9: graph, display function keyboard input (no echo),
1Ø: line printer output.

Others will be addid.

Transfer call is CSV 6,Ø; REQDES address in 4_W.

Physical handling is by interrupts if the current program's status is $\geq$ 12, by polling otherwise.

The Handler runs in supervisor mode; its status is 3 if working by interrupts, and equal to the current program's status if polling.

In user routine call: D is 6_W, R is 4_W. The routine runs in supervisor mode and has the Handler's status. Return by JMP Ø(T).

The stack is:

Ø - 1 :	return to the Handler,
2 - 3 :	the Handler's internal storage,
4 - 5 :	" " " "
6 - 7 :	link page address at Handler's start,
8 - 9 :	(Ø) " " "
1Ø - 1	(2) " " "
12 - 13:	(4) " " "
14 - 15:	(6) " " "
16 - 17:	return address,
18 - 19:	program status,
2Ø:	stack top at call or interrupt to Handler.

Transfer is compatible with DOST's wait-mode peripheral handling in that calls to them can be intermixed but not overlapped.

For *Drivers*, interrupt request level should be 3. DX is 6_W, BC is 3_B. - Normal "return" means JMP Ø(T). SAVE is to be called with CSC. (It may be serviceable to known that SAVE sets the left half of 7_B to the condition code as given in the interrupt vector, the rest of 6_W is clear.) Stack is (both after Driver call and after SAVE) as at user routine call without the topmost two words. Link page exists.

All presently existing Devices are driven by a single Driver: DOIT. It can be extended to handle other similar simple Devices. Other Drivers may use its conversion routines: INCODE and EXCODE. - For Code, DOIT interprets the characters of standard TPA peripherals in the obvious way except that "↑" is interpreted as "bar", "←" as "ɕ", "cr" as "LN", "alt mode" as "BØ" and "lf" as "Bl".

Items:

	for core	for virtual
INSYNT	CSV 6,2	CSV 6,3
INITEM	CSV 6,4	CSV 6,5
EXITEM	CSV 6,6	CSV 6,7

VALUE:$Ø_D$, FORMAT:6_B, TERM:7_B, LOC:4_W; SYNTAX:2_W.

There is a version to be called by JSR rather than CSV. Its starting address is the address of INSYNT-core, the others follow in steps of two.

*

Transfer and Item CSVs don't care about memory protection; they will be called by supervisor-mode "logistics" anyway. So the next "system" level ought to take care of it.